Eulerian Video Magnification With TBB

Using Multi-core Strategies to Improve the Processing with Eulerian Video Magnification

List of Authors (Grace Szpytman, Cherwa Vang)
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: gszpytman2@oakland.edu, cwvang@oakland.edu

***Abstract*—Eulerian Video Magnification (EVM) is an algorithm that magnifies subtle changes in color or movement. The EVM algorithm is computationally heavy, requiring convolutions, Fourier Transforms, and various math applied to each video frame. Since EVM is computationally heavy, it is traditionally run with prerecorded videos. To process a live video stream, the algorithm needs to run faster than the camera's frame rate. To decrease the computation time, efficient strategies must be applied to video processing to improve frame rate, decrease frame latency, and provide more capabilities. This paper aims to improve upon these features by taking advantage of multi-core embedded systems to divide the work between multiple cores.**

I. INTRODUCTION

The need for video processing and filtering has increased over the past decades. From phones, to vehicles, home security, other smart devices, and industrial uses, cameras are more and more involved in everyday embedded systems. Raw video footage isn't useful by itself. Often raw video footage undergoes various processing and filters to make the video footage usable in software. For example, sharpening filters to make the video clearer, and edge detection filters to recognize geometry. However, video processing and other filters can be computationally heavy. Some algorithms are so heavy that it's not practical to run the algorithm concurrently with a live video feed. Instead, the algorithm needs to run on prerecorded video. One way to decrease the computation time is to improve the processor.

Processors have improved over the years by increasing the clock frequency, improving the architecture, new technologies, and increasing the core count. By increasing the number of cores in a processor and designing software that takes advantage of a multicore processor, embedded systems can improve performance. To improve the performance of video in embedded systems, efficient strategies must be used to take advantage of multicore architecture. This paper will implement multiple strategies using Threaded Building Blocks (TBB) to optimize the video processing.

Eulerian Video Magnification (EVM) is an algorithm that magnifies subtle color and movement changes. This paper will focus on the motion magnification portion of EVM. The EVM motion magnification algorithm uses convolution filters, Fourie transforms, and various math to magnify subtle movements. The algorithm can be summarized with the following steps:

1. Fetch frame from webcam or video file
2. convert from RGB to LAB color space
3. Spatial decomposition with Gaussian then Laplacian pyramid
4. Temporal filtering to isolate the motion magnification
5. Amplify the motion and attenuate everything else
6. Reconstruct image and convert back from LAB color space to RGB color space.
7. Push frame to display or save video frame

These filters and image processing algorithms are computationally heavy. To improve the framerate, and take advantage of multicore processors, the EVM algorithm will be split into multiple stages, to run concurrently.

Three Parallelization strategies will be used in this paper:

- 3 stage pipeline. This strategy separates the calculation process from the fetch and store processes. The assumption with this strategy is that the fetch data and store data is a slow process. This is shown in Figure 1.

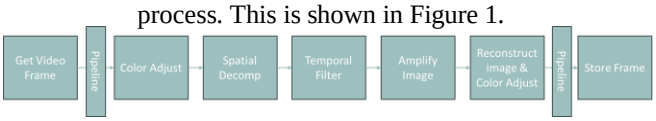

FIGURE 1: THREE STAGE PARALLEL PIPELINE

- 5 stage pipeline. This strategy tries to split the image processing time into 5 even stages. This will take advantage of processors that have multiple cores. This is shown in Figure 2.

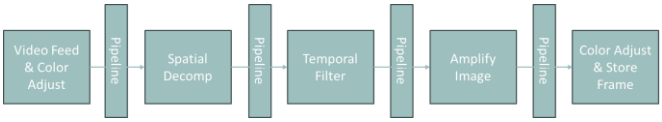

FIGURE 2: FIVE STAGE PARALLEL PIPELINE

- Parallel Reduce to parallelize the Temporal filter and amplify stages to process multiple filters in parallel. Then joining all the images into one to show the operator. This is shown in Figure 3.

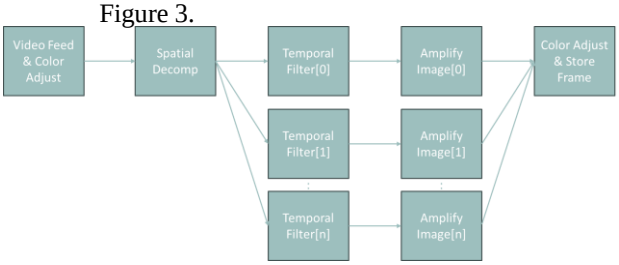

FIGURE 3: PARALLEL REDUCE MULTI FILTER PROCESS

The scope of the project will be to investigate the best way to split up the EVM algorithm to produce the fastest computation time or provide the best information to the operator.

EVM is useful in many applications such as vibration monitoring on an assembly line, or heartrate monitoring. Since EVM can magnify color or motion at various frequencies. The faster the processing, the higher the framerate. The frame rate is the same as the sampling rate, and a higher sampling rate will allow the algorithm to magnifier higher frequencies. E.g. To magnify a 60 bm heart rate, the camera will need to run at a frame rate of at least 120 frames per second or at a sampling rate of 120 Hz.

II. METHODOLOGY

As stated in the introduction and from Figure 1, the EVM algorithm consists of 7 stages. The first and last stages are image store and fetch stages and are not involved in the EVM algorithm. The middle five stages describe how the EVM algorithm works:

1. Convert from RGB to LAB color space
2. Spatial decomposition
3. Temporal filtering
4. Amplification and attenuation
5. Image reconstruction, color space conversion

A. Converting form RGB to LAB color space

Camera sensors and digital displays use the RGB color space to capture and display images. RGB stands for red blue and green values. The RGB color space stores the intensity of each color of each pixel to represent an image. RGB is good for digitally representing an image, however, LAB or YIQ color space more accurately represents how real eyes see color. LAB is a color space standard defined by the international Commission on Illumination (CIE). LAB stands for perceptual lightness, red-green perception, and blue-yellow perception [2]. LAB color space allows the image processing algorithm to manipulate the color values without affecting the brightness of the image. Figure 4 shows two images of a Rubik's cube, one image taken outdoors, and one outdoors [3]. The colors are separated to their RGB values. Notice how the color values differ between the two lighting conditions.

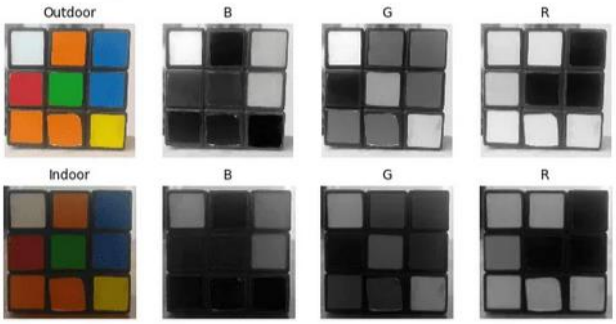

FIGURE 4: RGB COLOR SPACE AT DIFFERENT BRIGHTNESS

Figure 5 shows the same Rubik's cube in LAB color space. Notice that the Lightness value changes between the two lighting environments, but the A and B components are relatively the same. The color conversion is done using built in functions in Open Computer Vision (Open CV).

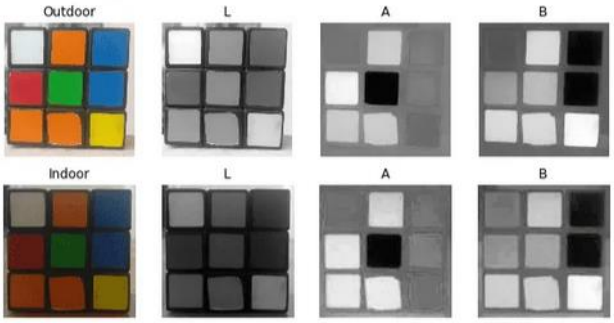

FIGURE 5: LAB COLOR SPACE AT DIFFERENT BRIGHTNESS

B. Spatial Decomposition

The purpose of spatial decomposition is to apply a filter that detects the edges of an image. The spatial decomposition starts by creating a Gaussian Pyramid. A Gaussian pyramid is created by applying a 5x5 gaussian filter to the image shown in Figure 6. The 5x5 Gaussian filter is a low pass filter that preserves low spatial frequencies.

$$G_k = \frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

FIGURE 6: 5X5 GAUSSIAN FILTER

Next the image is down sampled by 2 to create the second layer of the pyramid. This process repeats for as many layers as desired. This is shown in Figure 7.

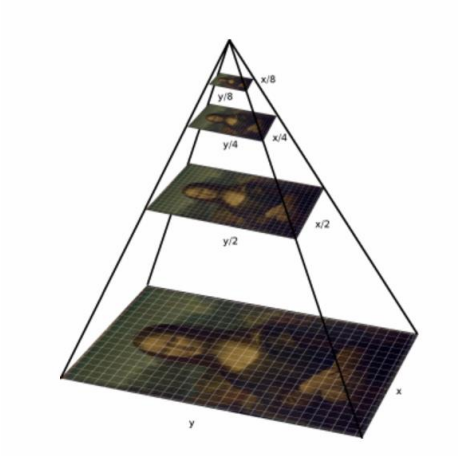

FIGURE 7: GAUSSIAN PYRAMID.

Once the Gaussian Pyramid is built, it is turned into a Laplacian Pyramid to detect edges. A Laplacian Pyramid is done by getting the difference between two adjacent Gaussian levels. This is done by taking the higher level, lower resolution images of the Gaussian Pyramid, and up sampling them. Then the difference between the up sampled image and the previous level is used to create a new pyramid. This is shown in Figure 8.

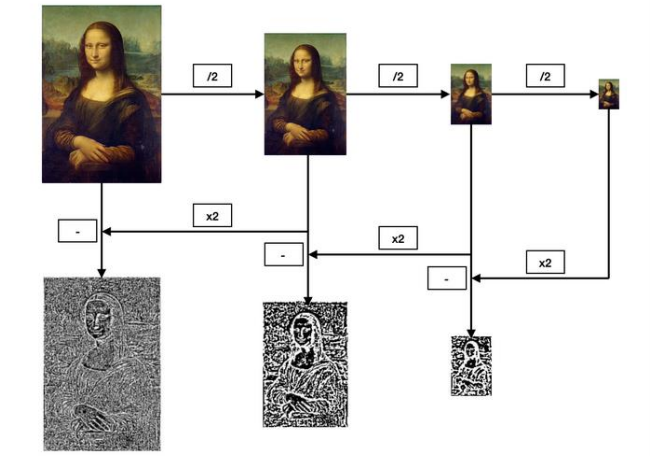

FIGURE 8: LAPLACIAN PYRAMID

C. *Temporal Filtering*

The purpose of Temporal Filtering is to keep the desired frequencies and filter out the undesired frequencies. Like the name suggests, the Temporal filter filters in time. In other words, the current frame relies on the previous frame to determine if the changes between those two frames happen within the desired frequency. The temporal filter is a first order Butterworth filter. First order is used over higher orders since motion magnification is not as uniform as color magnification. So, a filter that is tolerant to frequencies slightly outside the desired frequency is desirable. An example Butterworth filter is shown in Figure 9.

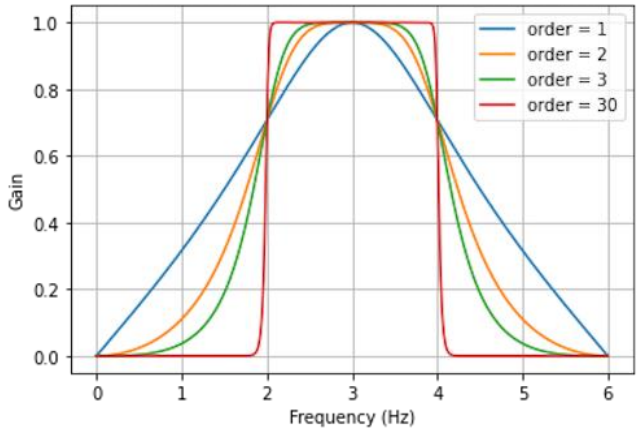

FIGURE 9: BUTTERWORTH FILTER

The Butterworth Filter equation is shown in Figure 10. The variables in Figure 10 are:

- y[n]: nth filtered image output of the current frame
- x[n]: nth Laplacian image output
- a0/a1: feedback coefficient
- b0/b1: feedforward coefficient

$$y[n] = \frac{1}{a_0}(b_0 x[n] + b_1 x[n-1] - a_1 y[n-1])$$

FIGURE 10: BUTTERWORTH FILTER

D. *Amplification and Attenuation*

After the image is filtered, the filtered frequencies are amplified. The amplification is done per pixel with the following equation shown in Figure 11. The variables in Figure 11 are:

- $M_{Fp}[L]$: Lth amplified output image of the current frame
- F_p : Current filtered pyramid from previous stage
- A: Attenuation Factor
- L: pyramid level
- λ : llamda, special wavelength
- δ : delta, displacement factor
- α_{new} : alpha, amplify level
- YIQ is another color format very similar to LAB

Eq 1 $$M_{F_p}[l] = \begin{cases} \alpha_l F_p[l] \text{ if Y component} \\ \alpha_l A F_p[l] \text{ if I or Q component} \end{cases}$$

Eq 2 $$\lambda = \sqrt{h^2 + w^2}$$

Eq 3 $$\delta(t) = \frac{\frac{\lambda_c}{8}}{1+\alpha}$$

Eq 4 $$\alpha_{new} = \frac{\frac{\lambda}{8}}{\delta(t)} - 1$$

Eq 5 $$\alpha_l = min(\alpha, \alpha_{new})$$

FIGURE 11: AMPLIFICATION EQUATION

As shown in Eq 1 in Figure 11, the Y and I/Q component of the image are amplified differently. The I/Q component may be attenuated by the variable A. This amplification is done at each level of the pyramid, except for the first and last levels.

E. Image Reconstruction & Color Space Conversion

Then each image in the Laplacian pyramid is combined by up sampling the higher levels images then adding them to the lower level images. This is repeated until the bottom and final layer is added. This is shown in Figure 12.

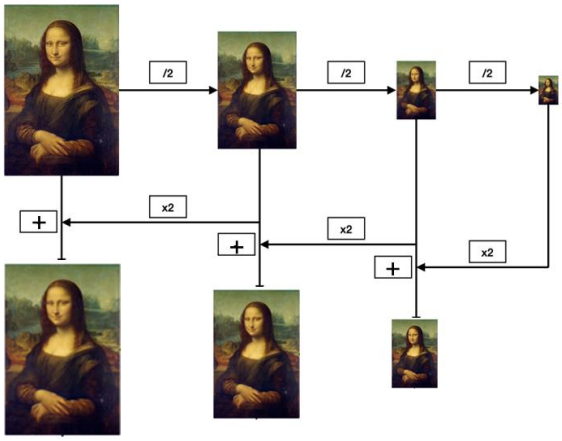

FIGURE 12: ADDING LAPLACIAN PYRAMID AFTER AMPLIFICATION AND ATTENUATION

After the image is rebuilt to the final level, the image is converted from LAB space back to RGB, and pushed to a video output, or saved to a file.

III. EXPERIMENTAL SETUP

This experiment used a high-performance laptop with a base processing speed of 1.9 GHz and included 8 cores and 16 logical processors as seen in Figure 13. The laptop was equipped with Ubuntu 24.04 which included libraries for multi-threading and TBB usage. This configuration was selected to best implement parallelization strategies.

Base speed:	1.90 GHz
Sockets:	1
Cores:	8
Logical processors:	16
Virtualization:	Enabled
L1 cache:	512 KB
L2 cache:	4.0 MB
L3 cache:	16.0 MB

FIGURE 13: PROCESSOR HW USED IN TESTING

As mentioned in the introduction, three parallelization strategies were utilized. A three stage TBB parallel pipeline was used to separate the calculations from the fetch and store process. Due to the temporal filter relying on previous frames, the parallel mode of the pipeline caused the code to hang and then a segment fault would be thrown. To rectify this the serial_in_order mode was used which allowed the incoming data to be calculated in order. The second strategy was a five-stage pipeline that would further break up the calculation stage into three stages. The image processing portion was separated into its individual calculations, the spatial decomposition, temporal filter, and image amplification. Due to the time and level of difficulty of the application a different approach was used. The final strategy implemented TBB parallel reduce, which parallelizes the temporal filter and amplify stages, to process multiple filters simultaneously before combining the results into a single image for the operator shown in Figure 3. However, due to parallel reduce class taking time to instantiate, the image processing took too long for the strategy to be practical for this application.

IV. RESULTS

After running each parallel implementation, they were compared against the sequential implementation. Figure 14 shows the average per frame calculation of the sequential run between two different image sizes.

Sequential (640x480)		Sequential (2280x3840)	
Trial	**Frame (ms)**	**Trial**	**Frame (ms)**
Trial1	19.92	**Trial1**	913
Trial2	14.94	**Trial2**	907
Trial3	19.92	**Trial3**	925
Trial4	17.94	**Trial4**	903
Trial5	16.95	**Trial5**	896
Average	17.93	**Average**	908.8

FIGURE 14: AVERAGE RUN TIME FOR SEQUENTIAL PROCESS

Figure 15 shows the average per frame calculation of the three pipeline run with two different image sizes.

3 Stage Pipeline (640x480)				
Trial	Get (us)	Calc (ms)	Set (ms)	Total (ms)
Trial1	176	6.7	33.53	40.41
Trial2	132	6.88	34.2	41.21
Trial3	196	6.87	34.7	41.77
Trial4	186	6.91	35.5	42.60
Trial5	152	6.58	33	39.73
Average	168.4	6.788	34.185	41.14

3 Stage Pipeline (2280x3840)				
Trial	Get (us)	Calc (ms)	Set (ms)	Total (ms)
Trial1	4.5	1050	10.1	1060.10
Trial2	4.2	1045	9.8	1054.80
Trial3	4.6	980	11.6	991.60
Trial4	3.9	955	9.9	964.90
Trial5	4.4	1011	10.4	1021.40
Average	4.32	1008.2	10.36	1018.56

FIGURE 15: AVERAGE RUN TIME FOR 3 STAGE PIPELINE

Looking at the results between the two lower resolution 640x480 images, the calculation times for the sequential operation is almost twice as fast as the 3 stage pipeline implementation (17.93 ms vs 41.14 ms). This is likely due to the overhead cost of creating the TBB Parallel Pipeline. However, when image size increases, the results are closer. When using the larger resolution 2280 x 3840 images, the sequential operation on average is only about 100 ms faster than the parallel process (908.8 ms vs 1018.56 ms).

One surprising result is how quickly the Get and Set stages are so fast. One of the assumptions made was that the process of fetching and storing each frame was slow relative to the calculation speeds. In certain cases this may be true, however, for our setup getting and setting data is fast.

Since the get and set are so fast, some of the calculations can be pushed to those stages. This 5 stage pipeline is shown in Figure 2. However, due to time constraints, the 5 stage pipeline design is currently not in a working condition. Below, Figure 16 shows what our theoretical timings would be with a 5 stage pipeline. These calculations were done by timestamping the 3 stage pipeline.

Trial	Get (us)	Spatial (ms)	Temp (ms)	Amp (ms)	Set (ms)	Total (ms)
Trial1	4	360	409	245	33	1047.00

FIGURE 16: THEORETICAL TIMINGS OF A 5 STAGE PIPELINE

Had the 5 stage pipeline work as intended, then although the total time per frame is 1047 ms, however, since this is a 5 stage pipeline, five frames could be calculated simultaneously. This makes the throughput of the 5 stage pipeline process ideally 5 times faster. However, since the load balancing of each stage isn't perfect, the actual performance increase is limited by the slowest stage. From Figure 16, the slowest stage is the temporal filtering stage at 409 ms. This means that after 5 stages of latency, each new frame will come out at approximately 409 ms. Our theoretical timings in Figure 16 doesn't take into account the pipeline overhead time. Comparing the pipeline overhead between the sequential and 3 stage pipeline, the setup cost of creating the two additional stages to the pipeline is ~100 ms. If we assume that the adding two additional stages also adds about ~100 ms of overhead, then the theoretical throughput of the 5 stage pipeline is about 1 frame per 500 ms. The performance increase is shown in Figure 17, with the five stage pipeline having a theoretical performance increase of 1.82.

	Longest Stage (ms)	Performance Increaes
1	908.8	1
3	1018.56	0.89
5	500	1.82

FIGURE 17: PERFORMANCE INCREASE WITH PIPELINE

Looking at the results, the 3 stage pipeline is slower than the sequential one. The 3 stage pipeline would be faster had the image processing been divided more evenly. The theory was that getting a frame, and saving it to memory is slow compared to the image processing time. From these results, the assumption is wrong. An improvement to be made to the 3 stage pipeline would be to move some of the image processing to the get and set stages of the 3 stage pipeline. The theoretical improvements should be between the sequential and five stage pipeline.

Decreasing the calculation time it takes to do image processing on a frame is one way to increase performance. Another way of increasing performance is to give more information to the operator so that the operator can make a more informed decision. This was the thought process behind using parallel reduce to increase performance. The goal was to have multiple parallel paths that filtered and amplified different frequencies at the same time. After splitting to perform their individual filtering and amplification, the join process will concatenate the various different filtered images into one large image. This was the goal, however, after running with only one filter (parallel reduce with only one path, in other words, no parallelization), the setup time required by creating the parallel reduce class is so heavy, that it is not practical to use. This is shown in Figure 18.

Parallel Reduce (640x480)	
Trial	**Calc (s)**
Trial1	1.04
Trial2	1.1
Trial3	1.16
Trial4	1.1
Trial5	1.07
Average	1.094

FIGURE 18: AVERAGE RUN TIME OF PARALLEL REDUCE IMPLEMENTATION

They are many reasons why the parallel reduce implementation may not show a speed improvement. The most likely reason why the implementation is slower than the sequential implementation is likely due to the creation of the class. For each filter, a new class needs to be created, and the process of creating that class is slow. This can be seen in how the parallel reduce was implemented in Figure 19.

```
gettimeofday(&start, NULL);
while(inputFrameParameter.frameNum >= 0){
  //get frame
  inputFrameParameter = GI();
  printf("*** frameNum = %i\r\n", inputFrameParameter.frameNum);
  //create new class
  EulerianMotionMag EMM(
                        input_width, input_height, levels,
                        cutoff_freq_low, cutoff_freq_high, lambda_c, alpha,
                        chrom_attenuation, exaggeration_factor, delta, lambda);
  parallel_reduce(blocked_range<size_t> (0,parallelFilterCount), EMM);

  //save data
  SO(EMM.img_input_);
};
gettimeofday (&end, NULL);
```

FIGURE 19: CODE SNIP OF EMM CLASS CREATION

V. CONCLUSIONS

The results of the experiment shows that there is potential for improving the performance of Eulerian Video Magnification. By adding more pipelines and spreading out the calculations over different stages will increase the throughput at the cost of increasing the latency. The image processing improvement is more apparent when processing large resolution videos. With a potential gain of 1.82 from our theoretical results in our 5 stage pipeline. If time had permitted, the bugs in the 5 stage pipeline would've been resolved, and empirical results could've been achieved.

There is still potential in the parallel reduce implementation. Due to time constraints, the bug that caused the parallel reduce to decrease performance could've been patched. An improvement on the algorithm doesn't necessarily need to be an improvement to the calculation times. Improvement can also be in the form of providing more information so that the operator can make an informed decision. For example, a video feed that shows motion magnification at 8 different frequencies.

VI. REFERENCES

[1] H. B. Belgacem, "Eulerian video magnification," Eulerian Video Magnification · Hussem Ben Belgacem, https://hbenbel.github.io/blog/evm/

[2] Gupta, V. (2024) *Color spaces in opencv (c++/python)*, *LearnOpenCV*. Available at: https://learnopencv.com/color-spaces-in-opencv-cpp-python/

[3] International Color Consortium (2004) *Specification ICC.1:2004-10*. Available at: https://www.color.org/icc1v42.pdf